EXAMEN – ARCHITECTURE DES DONNÉES & IA

Durée : 1 :30min | Documents : Tous autorisés (Internet, cours, LLM)

Consignes importantes

- Toute réponse doit être :
  - ***Justifiée / Contextualisée / Argumentée techniquement***
    - ✓ **Ce que c'est** : Ne pas simplement dire "il faut utiliser Polars", mais expliquer *pourquoi* dans ce cas précis (taille mémoire 16Go, format Parquet existant, besoin de lazy evaluation).
    - ✓ **Piège anti-LLM** : ChatGPT donne des réponses génériques. Toi, tu dois adapter à l'énoncé spécifique (ex: "ici, vu que le fichier fait 12Go et que nous avons une contrainte RAM...").
- Les réponses **génériques ou copiées** seront pénalisées (**jusqu'à -50%**)
  - ✓ Le piège : Coller une définition Wikipédia de "Git rebase" alors que la question demande une procédure précise de suppression de mot de passe dans un historique déjà pushé.
  - ✓ L'enjeu : Tester la compréhension, pas la capacité à copier. Un LLM ne sait pas que dans ce scénario précis, git revert ne suffit pas car le mot de passe reste dans l'historique.
- Les calculs doivent inclure :
  - ***Hypothèses / Ordres de grandeur***
    - ✓ **Pourquoi** : Un bon architecte data raisonne en mémoire, pas juste en syntaxe.

- ✓ **Exemple** : "Pandas charge 12Go CSV → expansion mémoire ×3 (objets Python) = 36Go > RAM 16Go → crash. Hypothèse : strings UTF-8 lourdes. Ordre de grandeur : Polars réduit à ~8Go grâce aux types Arrow compacts."
- ✓ **Anti-LLM** : Les modèles hallucinent les chiffres. Toi, tu montres le raisonnement : 10M lignes × 50 colonnes × 8 bytes (int64) ≈ 4Go brut, plus overhead...

- L'objectif n'est pas de trouver une réponse, mais de démontrer :
  - ***votre raisonnement***
  - ***votre compréhension des systèmes data***

Ces consignes obligent à penser plutôt qu'à recopier, ce que les LLM ne font pas encore bien seuls dans un contexte spécifique.

PARTIE I – DIAGNOSTIC ARCHITECTURAL (30 points)

Exercice 1.1 – L'Antipattern du Data Pipeline (15 points)

Votre collègue débutant propose le code suivant pour traiter 50 millions de lignes de logs clients. **Analysez les 3 erreurs critiques** (performance, mémoire, architecture) et proposez une solution refactorisée.

```python
# pipeline_errone.py
import pandas as pd
def process_logs():
    df = pd.read_csv("logs_50M.csv")  # Fichier de 12Go
    for i in range(len(df)):
        if df.iloc[i]['status'] == 'error':
            df.iloc[i]['status'] = 'failed'
    df['json_meta'] = df['metadata'].apply(lambda x: eval(x))
    df.to_csv("output_final.csv")
    return df
```

Consignes :

- Identifiez la complexité algorithmique de la boucle for et son impact sur le temps de traitement
- Expliquez le risque sécuritaire de eval() et proposez l'alternative sécurisée pour parser du JSON
- Réécrivez le pipeline en utilisant soit Pandas vectorisé, soit Polars lazy evaluation

- **Argumentation anti-LLM :** Pourquoi, même avec Polars, la lecture d'un CSV 12Go sur une machine 16Go RAM échouera-t-elle sans optimisation supplémentaire ? (indice : dtype inference)

Exercice 1.2 – Git : Gestion de Crise & Historique (15 points)

Scénario : Vous découvrez que le commit a1b2c3d (il y a 3 jours) contient un mot de passe en clair dans config.ini. Entre-temps, des collègues ont pushé des commits e4f5g6h et i7j8k9l sur main.

Questions :

1. Pourquoi git revert a1b2c3d est-il insuffisant pour effacer le mot de passe de l'historique Git ? Expliquez la différence entre "annuler un changement" et "effacer l'historique".
2. Décrivez la procédure précise utilisant git rebase -i ou git filter-repo pour supprimer définitivement le fichier sensible de l'historique tout en conservant les modifications légitimes des commits suivants.
3. Vos collègues ayant déjà pullé l'historique contaminé, quelle commande doivent-ils exécuter pour synchroniser proprement après votre rewrite ? Pourquoi git pull simple créerait-il un désastre ?
4. Proposez une règle de prévention (GitHub Action) qui bloquerait automatiquement un push contenant des patterns de secrets (regex pour mot de passe, API key).

PARTIE II – ARCHITECTURE DE STOCKAGE & MODÉLISATION (35 points)

Exercice 2.1 – Design de Stockage Colonne & Partitionnement (15 points)

Vous devez architecturer le stockage des 50M logs (12Go CSV) en format Parquet pour des analyses IA fréquentes. Contexte : requêtes types filtrent sur date (derniers 30 jours) et user_id, agrégations sur montant.

Questions d'architecture :

1. **Stratégie de partitionnement Hive :**
   Proposez une structure de dossiers bucket/date=.../user_segment=.... Justifiez le choix de la granularité (daily vs monthly) et expliquez pourquoi partitionner sur user_id directement serait une erreur (cardinalité trop élevée).
2. **Optimisation des row groups :**
   Sachant que Polars lit les row groups entiers en mémoire, calculez la taille optimale d'un row group pour une machine 16Go RAM (règle : 1 row group ≤ 20% de la RAM). Combien de row groups contiendra votre fichier final ?

3. **Choix du codec :**
   Comparez Snappy (décompression rapide, ratio moyen) vs Zstandard (compression forte, décompression plus lente). Dans quel cas de lecture (analyse ponctuelle vs scan complet) privilégieriez-vous chacun ? Justifiez par l'architecture CPU vs I/O.

4. **Schéma Arrow & Types :**
   Le CSV contient des colonnes avec valeurs manquantes. Expliquez pourquoi utiliser string pour tout est inefficace, et proposez l'utilisation de types nullable (int64[pyarrow]) pour optimiser la compression et la vitesse de lecture.

Exercice 2.2 – Polars : Requêtes Complexes & Optimisation (20 points)

Vous devez implémenter une jointure complexe avec gestion mémoire stricte :

Contexte :

- transactions : 10M lignes, 50 colonnes, partitionné Parquet par date
- clients : 500k lignes, 200 colonnes (150 inutiles), format CSV

Code problématique :

```
import pandas as pd
trans = pd.read_parquet("transactions/")
clients = pd.read_csv("clients.csv")
merged = trans.merge(clients, on="client_id")  # MemoryError ici
result = merged.groupby("segment")["montant"].sum()
```

Tâches :

1. **Analyse mémoire (5 points) :**
   Calculez l'empreinte mémoire approximative :
   - Transactions : 10M × 50 cols × overhead pandas
   - Clients : 500k × 200 cols
     Pourquoi le résultat dépasse-t-il les 16Go RAM même après la jointure ?
     (indice : fragmentation mémoire, copies internes)

2. **Refactoring Polars Lazy (10 points) :**
   Réécrivez en Polars avec :
   - **scan_parquet()** et **scan_csv()** (lazy)
   - Projection pushdown : sélectionnez uniquement les 50 colonnes nécessaires de clients avant la jointure

- Predicate pushdown : filtrez transactions sur les 6 derniers mois avant la jointure
- streaming=True pour le groupby final
- Utilisation de join() avec how="inner" et stratégie de broadcast (la petite table clients est broadcastée, pas partitionnée)

3. **Question d'architecture (5 points) :**
   Malgré l'optimisation Polars, expliquez pourquoi dans une architecture de production distribuée (Spark/DuckDB), il serait préférable de pré-calculer une table agrégée quotidienne plutôt que de faire cette jointure à la volée. Utilisez le concept de "**data locality**" et le coût du "**shuffle**" réseau dans votre argumentation.

PARTIE III – PIPELINE HYBRIDE & INTÉGRATION (25 points)

Exercice 3.1 – Chaîne de Traitement Multi-Types (25 points)

Créez un pipeline ETL complet gérant les 4 types de données du cours :

Données sources :

- **Structuré** : commandes.csv (client_id, montant, date, ville)
- **Semi-structuré** : events.jsonl (logs JSON ligne par ligne, nested)
- **Non-structuré** : Dossier invoices/ avec images JPG des factures (simuler l'OCR avec une fonction mock : mock_ocr(path) → str)
- **Vectoriel** : embeddings.npy (matrice NumPy 100k × 768, float32)

Contraintes architecturales :

1. **Chargement polymorphe :**
   Implémentez une fonction load_data(source_path) qui détecte automatiquement le type (extension + inspection contenu) et retourne un objet unifié (DataFrame Polars pour tous les types, avec métadonnées de type).
2. **Normalisation :**
   - Convertissez les JSON nested en structure aplatie (flatten) avec notation pointée (metadata.device.type → colonne metadata_device_type)
   - Intégrez le texte OCR extrait des images comme nouvelle colonne invoice_text jointe aux commandes par order_id
3. **Gestion de la mémoire :**
   Les embeddings 100k×768 occupent ~300Mo en RAM. Chargez-les par batch de 10k

lignes pour calculer la similarité cosinus avec des vecteurs requête, sans jamais charger la matrice entière en mémoire vive simultanément.

4. **Export architecturé :**
   Sauvegardez le résultat final en Parquet avec :
   - Partitionnement Hive par ville et date
   - Compression Zstandard niveau 3
   - Métadonnées dans le footer Parquet indiquant le nombre de valeurs nulles par colonne
5. **Traçabilité :**
   Générez un fichier manifest.json à côté des données finales contenant :
   - Hash SHA256 des fichiers sources
   - Version de Polars utilisée
   - Timestamp et nombre de lignes traitées

PARTIE IV – RÉFLEXION ARCHITECTURALE (10 points)

Question de synthèse : Trade-offs Architecturaux (10 points)

Scénario : Vous devez concevoir l'architecture data pour une plateforme de recommandation IA temps réel :

Option A (Lakehouse Moderne) :

- Stockage : Delta Lake sur S3 (format Parquet avec transaction log)
- Traitement : Polars/DuckDB pour batch, API Python pour streaming
- Coût : Faible (stockage objet), latence batch : minutes
- Risque : "Small file problem" si micro-batches trop fréquents

Option B (Warehouse Classique) :

- Stockage : Snowflake/BigQuery (colonne optimisé, index automatiques)
- Traitement : SQL pur, maintenance nulle
- Coût : Élevé (compute dédié), latence : secondes
- Avantage : ACID strict, time-travel natif

Questions :

1. **(4 points)** Expliquez le "**small file problem**" de l'Option A : pourquoi créer des fichiers Parquet de 5Mo toutes les minutes finit par tuer les performances de lecture, et comment le résoudre (compaction, coalescing) sans perdre la fraîcheur des données.
2. **(3 points)** Pourquoi l'utilisation de Git-LFS pour versionner les jeux de données d'entraînement (plusieurs Go) est une **anti-pattern architectural** comparé à l'utilisation de DVC (Data Version Control) ou d'un simple stockage versionné S3 avec références dans Git ?
3. **(3 points)** Proposez une stratégie de "**data quality gates**" implémentable dans GitHub Actions qui :
   - Valide le schéma des Parquet générés (colonnes obligatoires présentes)
   - Vérifie la distribution statistique d'une colonne clé (pas de drift > 20% par rapport à la baseline)
   - Bloque le merge sur main si les checks échouent
     Indice : utilisez pandera ou great_expectations en ligne de commande.

